Alarm Scanning for PPC

Improve performance

Tue, May 1, 2001

The alarm scan logic used in IRM software checks every analog channel and every digital bit for possible alarm checking, even though most channels and bits don't require it. In IRMs, nonvolatile memory access time is not significantly different from dynamic RAM access time. But in the PPC systems, nonvolatile memory access is much slower than DRAM access. This note describes a scheme to improve alarm scan timing for PPC nodes.

Background

As a reference point, the time for an IRM to perform a complete alarm scan, which it does every 15 Hz cycle, is about 1.0–1.5 ms. The current time for PPC systems is perhaps 1.5–2.0 ms. Both of these times are for systems that allocate 1K channels and 1K bits. For the Linac upgrade, however, it is planned to double the number of channels allocated in some node, and perhaps also double the number of bits. This could double the alarm scan time so that it could approach 4 ms. This is the time we are trying to reduce. Whether this is deemed significant enough to matter, for a job that is performed once every 66 ms, is a separate consideration.

The plan

The basic idea is to take advantage of the fact that the vast majority of channels and bits in any system are *not* enabled for alarm scanning, so that if we could merely check alarms for those channels and bits that *are* enabled for alarm scanning, the time required could be much less.

The plan is to occasionally perform a scan that builds up a list of pointers to active channel and bits. The pointers would point to `ADATA` table entries for each active channel and to `BALRM` entries for each active bit. It may also be necessary to keep a copy of the channel number or bit number of the entry to which each pointer refers. The alarm scanning logic would be modified to follow these pointer entries to check only those channels and bits that need to be checked. It would completely ignore the rest.

Thinking out loud

We must decide how often it is worth reconstructing this alarm pointer table. The time required to construct the table may be comparable to the present alarm scan times. It would probably be done all at one time so that the time we are trying to reduce would be required during a single 15 Hz cycle, so that that particular cycle would not save time.

Ideally, we would not have to reconstruct the table unless a change is made to the alarm flags such that the Active bit becomes set. Such changes are very infrequent in normal operation. If we assume that only listype-directed processing would set an Active flag, and not a memory access, then special code could monitor whether reconstructing the pointer list was needed and set a flag that would alert the `Alarms` task its next time around. One might imagine merely adding another pointer entry to the list to satisfy an additional channel or bit to be scanned, rather than rebuilding the entire array of pointers. Such an addition would result in an immediate change in the alarm scan, which is good. Turning off the Active bit should also result in an immediate change, because a pointer might point to what is now an inactive entry, and it would not check that entry. Still, it may be useful to occasionally reconstruct the pointer array just to be safe. If the alarm flags word were modified in memory, which is very unusual, at least the list of active entries would not remain invalid forever.

When a setting is done that specifies the alarm flags listype or the nominal value listype with 6 bytes of setting data in tow, a special routine is called. The routine that is called for setting the analog alarm flags is `SETANOM`. The routine required for setting the binary alarm flags word is simply `SETAFLG`. (`SETANOM` calls `SETAFLG`.) So, the special code for monitoring when alarm flags are changed can be placed in `SETAFLG`.

Since the `SETAFLG` logic preserves the active bit for one more alarm scan in the case that it is to be turned off, the reconstruction operation logic should be placed at the end of the alarm scan, not at its beginning. If the bit is being turned off, the Active bit will be forced set, and another bit will be set that will cause the alarm scan logic to remove the Active bit, once it has had a chance to determine whether a "good" message should be posted. The new action by `SETAFLG` should merely trigger a reconstruction of the fast alarm scan table, at the conclusion of the next alarm scan.

The reconstruction effort can be scheduled separately for the analog and digital cases. If only a change in an analog channel's active bit is done, only the analog reconstruction should be required.

The case of the comment alarms might also be abbreviated. We have a default table size of 64 entries, whereas we have never used more than 2 entries. A reconstruction scan can also be done for these entries. We need to add some new set-type code to watch for changes in the comment alarm flags so that such changes will force reconstruction.

The set-type routines need to signal the `Alarms` task to do the reconstruction. The easiest way to do this is by sending an event to the `Alarms` task. Three separate event bits could be used to cover the three basic cases of analog, binary, and comment.

The layout of the structure needed to perform the fast alarm scan is an array of offsets to `ADATA` entries for which the Active alarm flags bit is set, or it may simply be an array of channel numbers, which are trivially related to the offset.

The logic, then, merely involves sequencing through the array of channel numbers, obtaining a pointer to the `ADATA` entry for that channel, and continuing with the usual logic: check for the Active bit, and if set--which it will very likely be--branch to the usual code for processing, after which the code returns to the top of the loop to pick up the next entry in the array of active channel numbers.

Similar logic can be used for the analog, binary and comment cases, using the appropriate array of indexes for each case.

After all alarm scanning is complete, check for the three additional events that might be set to enable reconstruction of the appropriate array of table indexes. These events will not be waited on, but rather watched for following the 15 Hz alarm scan.

Where should the memory be found for holding the index arrays? It may be useful to allocate a block of memory for each of the three cases for this purpose. Upon initialization of the `Alarms` task, these pointer variables will be set to `NULL`. For processing each case, if the pointer is `NULL`, skip the alarm scan for that case, but allocate the block instead and set the event bit. After the alarm scan logic, check for the appropriate event bit set. If the event was set, construct the array of indices according to which table entries have the Active bit set in the alarm flags field. During the next cycle, the valid pointer will enable the alarm scan to be performed.

By using allocated memory, we have to work a bit to find the addresses of the arrays being scanned for alarms, but that should not be a problem. Just find the `Alarms` task variables, and the three pointers should be there.

Post-implementation

The implementation of this new logic was done along the lines described above. Here are more of the details that were fleshed out during implementation.

Each of the three allocated active record blocks uses the same header structure that provides some simple diagnostics.

Field	*Size*	*Meaning*
`MBlkSize`	2	Size of allocated block
`MBlkSpar`	4	(spare words)
`MBlkType`	2	Memory block type# =`000A`
`ActivT`	2	Time to perform reconstruction of active entry array in us

`ActivN`	2	Current #active entries
`ActivMn`	2	`ActivN` minimum
`ActivMx`	2	`ActivN` maximum
`ActivC`	4	Count of alarm-flag-change-initiated reconstructions
`ActivCB`	4	Count of periodic backup reconstructions
`ActivDT`	8	BCD Date/time of last non-backup reconstruction
`ActivA`	2048	Array of active entries (assuming 1024 allocated entries)

By looking up the `Alarms` task variables in the usual way, one can find the pointers to the three active entry blocks, for analog channels, binary bits, and comments. The diagnostics show how many times a reconstruction takes place, especially when it is a result of an alarm flags Active bit being changed. It also shows the time that such a change was last done. In any case, it includes the time for performing the reconstruction.

The initial implementation is for the 68040-based PowerPC. In a system that has 65 active analog channels and one active bits and 2 comments, the execution time of the `Alarms` task is about 400 us. This same implementation was also tried on a 68020-based system that has 4 active analog channels, 1 active bit, and 1 comment. This reduced that node's `Alarms` task time from 8 ms to 0.5 ms. (Its timing is only available with half-ms resolution.) Although the old Linac systems would benefit from this implementation, since they typically take about 8 ms to perform their alarm scans, we are currently upgrading those systems to the PowerPC version, so they will not likely use this latest version of the system code.

The expected benefit in using this new logic should be quite dramatic for the PowerPC case. Running at 233 MHz cpu clock, the access time to the non-cacheable nonvolatile memory is about 1 us. This was the motivation for considering the new logic, in which only the entries that need to be examined are actually checked for alarm conditions.

One may have considered a different approach, that of copying the `ADATA` table into ordinary DRAM, where the access time is very fast. But keeping these two sets of entries in sync would have been difficult. An alarm scan can cause several bits in an alarm flags word to change. If the system dies, we need the nonvolatility of the entire table, including the alarm flags, settings, nominal and tolerances. This approach seems less risky, where the actual nonvolatile entries are still scanned for alarms. The performance improvement arises from skipping all those entries whose Active bits are off.

More details

Each change in an alarm flags word uses a special set-type routine invoked via the relevant listype. This routine determines whether the alarm flags Active bit is being changed. If it is, it signals the `Alarms` task via the appropriate event. Although the `Alarms` task is not actually waiting on these events, it will detect them after its next alarm scan. This is done because a user who tries to turn off the Active bit will cause the "bypass control" bit to be set and the Active bit to remain set, so the next alarm scan has a chance to detect this case, and if necessary, emit a "good" alarm message.

In addition to a reconstruction resulting from a change in an Active bit, it is forced to occur periodically, just as a precaution. The period as implemented is about 9 minutes, or 4096/15 seconds at 15 Hz. All 3 reconstructions take place periodically, but they do not occur on the same 15 Hz cycle.

The modifications to the `ASCAN`, `BSCAN`, and `CSCAN` routines in the `Alarms` task are made as much alike as possible. Two new routines were added. The `ACTBLOCK` routine allocates an active index block, and the `ACTBUILD` routine reconstructs the block.

Minimize accesses to nonvolatile memory

Since the nonvolatile access time is so slow, we need to look at means of limiting the required accesses to improve the time for alarm checking. This is especially true for the `ASCAN` routine that accesses entries in the `ADATA` table.

Fields of an `ADATA` table entry that are accessed for alarm scanning are only a few:

```
Reading
Nominal
Tolerance
Flags
Count
```

The `Flags` field must be looked at every time. The `Count` field only needs to be accessed when alarm changes are detected. The `Reading`, `Nominal` and `Tolerance` fields need to be checked every time, of course. We may modify the `Flags` and/or `Count` words, but none of the other fields. We can read the `Reading` word separately, then read the `Nominal` and `Tolerance` words using a single long word access. This will save the time needed for one access every time. We can also read the `Flags` and `Count` words as a single long integer, also saving one access some of the time. (It may not be necessary to examine the `Count` word in many cases.) We may have to modify one or both of them, but often we only have to read it.

Examples from the present code:

Near the end of the `ASCAN` loop, there is a place where the #times nibble, which is part of the `Count` field, is to be reset. If we have already a copy made of this `Count` field, we should first see whether the nibble is already set, before updating the `Count` word. Also, with a copy of that field already handy, we do not have to both read and write it to clear the nibble, but can rather write the updated `Count` value with the hi nibble cleared. (We already know the value of the low 12 bits.)

New viewpoint:

How about this? Copy out the `Flags` and `Count` words as a long word, then separate them into separate local variables for the flags word and the count word. Any place in the code used in checking for alarms where this information needs to be accessed or modified, do it to these local copies. At the end of the loop, check whether any changes have been made to any of the potentially-alterable fields. If the final value of the local flags word is different from the original value, or if the count value has changed, perform the appropriate long integer write to update both the `Flags` and `Count` fields.

Delay accessing the `Reading`, `Nominal` and `Tolerance` fields until we know that these fields need to be referenced, based upon the floating point flag bit in `Flags`. Only then do we know whether the `ADATA` table needs to be accessed for these fields, or whether it is instead the `FDATA` table that should be accessed. Both tables are in nonvolatile memory. For `ADATA`, one can access the `Nominal` and `Tolerance` words as a single long word. For `FDATA`, two 32-bit accesses are required to get the two floating point fields.

A simple example can illustrate the potential savings of time. For the integer case of a reading that is currently in the "good" alarm state and remains so, there are now 7 accesses to nonvolatile memory. In principle, there might be only 3. These would be the `Flags/Count` long word, the `Reading` word, and the `Nominal/Tolerance` long word. In this simple case, neither the `Flags` word nor the `Count` word needs to change. In cases where the tries nibble needs to be changed, say, there would be only one more access required.

In some of the Linac nodes, there may be 150 readings checked, including both analog and binary. If the time required was 7 times this amount, it would cost 1 ms, where less than half that amount might do the job.

If the code were modified according to this scheme, and later on, a faster form of nonvolatile memory were found, the code would still run faster, maybe almost 3 us faster than before. A complete alarm scan done in well under 1 ms is ok for a 66 ms cycle time.

Again, the current time required in a PPC node with 45 signals is about 2.2 ms. This should be reducible to well under 0.5 ms with the changes outlined here.

The alarm scan code is fairly neatly partitioned. The `ASCAN`, `BSCAN`, or `CSCAN` routines perform the basic scan, invoking a routine such as `ASNEW` if there is a change to report. That routine calls `ANEW`, which in turn calls `GETABLK`, `ADDTIME`, and `QMSG`. It should be straightforward to find all the references to the `ADATA` fields `Flags` and `Count`, or the same fields used in the binary and comment cases. Most all of the references in question occur in the basic scan code, but a few occur in routines that it calls.

The basic idea is to let these few routines work with temporary copies of the nonvolatile fields. At the end of the `ASCAN` loop, say, a check is made to determine whether the `Flags` and `Count` fields have been modified; if they have, then write to both fields (using a long word write) with the assembled local copies of flags and count.

Knowing that each reference to nonvolatile memory might require 200 wait cycles, in CPU cycle terms, can have a major impact on how the code should be organized. Local variable accesses, since they are in faster cacheable dynamic ram, almost don't count. The time needed for intensive use of nonvolatile tables depends almost entirely on the number of accesses made to nonvolatile memory. An algorithm that minimizes the number of such accesses will win.